Annals of Operations Research 115, 51–72, 2002

Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search

STEVEN PRESTWICH s.prestwich@cs.ucc.ie
Cork Constraint Computation Centre, Department of Computer Science, University College, Cork, Ireland

Abstract. Systematic backtracking is used in many constraint solvers and combinatorial optimisation algorithms. It is complete and can be combined with powerful search pruning techniques such as branch-and-bound, constraint propagation and dynamic variable ordering. However, it often scales poorly to large problems. Local search is incomplete, and has the additional drawback that it cannot exploit pruning techniques, making it uncompetitive on some problems. Nevertheless its scalability makes it superior for many large applications. This paper describes a hybrid approach called Incomplete Dynamic Backtracking, a very flexible form of backtracking that sacrifices completeness to achieve the scalability of local search. It is combined with forward checking and dynamic variable ordering, and evaluated on three combinatorial problems: on the n-queens problem it out-performs the best local search algorithms; it finds large optimal Golomb rulers much more quickly than a constraint-based backtracker, and better rulers than a genetic algorithm; and on benchmark graphs it finds larger cliques than almost all other tested algorithms. We argue that this form of backtracking is actually local search in a space of consistent partial assignments, offering a generic way of combining standard pruning techniques with local search.

Keywords: hybrid search, maximum cliques, Golomb rulers, n-queens

1. Introduction

Systematic backtracking has been applied to combinatorial problems for several decades. Backtracking algorithms have the considerable advantage of *completeness*: if there is a solution then they will find it; they can be used to enumerate all solutions; and if there is no solution then they are able to report the fact. For optimisation problems they are guaranteed to find optimal solutions, and can prove them optimal by failing to find better solutions. A further advantage of backtracking is that it can be combined with powerful search tree pruning techniques such as branch-and-bound, constraint propagation and dynamic variable ordering. A drawback of backtracking is that it sometimes scales poorly to large problem instances: a choice made high in the search tree may lead to a dead-end, from which the algorithm may take a very long time to recover. A great deal of research has been devoted to improving the scalability of backtrackers, resulting in what are sometimes called *intelligent backtracking* algorithms. Most of these are related to standard *chronological* backtracking, but are able to jump back to higher nodes in the tree, thus eliminating entire subtrees while preserving completeness. A particularly interesting example is Dynamic Backtracking (DB) [15], which is able to backtrack to a variable without removing the intervening assignments, effectively reorganising the

search tree dynamically. However, though often successful, intelligent backtracking has its own dangers. For example DB is no better than chronological backtracking on the n-queens problem, only sometimes better on graph colouring problems [26] and much worse on random 3-SAT (though a modified version is no worse) [3].

A significant discovery of the 1990s was that some hard combinatorial problems can be solved much more quickly by *local search* than by backtracking. Backtrackers are able to solve n-queens problems with not much more than 100 queens, and random 3-SAT problems with a few hundred variables; in contrast, local search can efficiently solve problems with millions of queens, and random 3-SAT problems with thousands of variables. Unlike backtrackers, local search algorithms typically assign values to all variables, then attempt to remove constraint violations by changing assignments (either randomly or by focusing on those causing violations), a technique sometimes called *repair*. Early examples are the Min-Conflicts [30] and Breakout [32] algorithms for constraint satisfaction problems, and the GSAT [43] and other [20] algorithms for satisfiability problems. Local search is usually incomplete, but very useful for applications in which we simply wish to find a solution quickly. It is a special case of the more general class of *stochastic search* algorithms, which includes genetic algorithms, simulated annealing and neural networks. Unfortunately, most local search algorithms have a drawback besides that of incompleteness: they do not exploit the powerful pruning techniques available to backtrackers. Min-conflicts was found to perform poorly on crossword puzzles and some graph colouring problems [26], while GSAT and other more recent local search algorithms for SAT are easily beaten by backtrackers on problems such as quasigroup existence [50]. This makes local search unsuitable for certain problems, typically (though not always) those with a great deal of structure and few solutions.

Hence neither backtracking nor local search is ideal for problems that are both large and highly structured. This situation has motivated research into the design of hybrid algorithms combining features of both types of algorithm. One such hybrid is Partial Order Dynamic Backtracking (PDB) [16], which aims to improve the scalability of DB without sacrificing completeness. Based on the intuition that poor scalability is caused by inflexibility in the choice of backtracking variable, PDB allows greater flexibility than DB. Another hybrid approach is to use a systematic backtracker in a non-systematic way. Iterative Sampling [28] restarts a constructive search every time a dead-end is reached, using randomised heuristics. Variations on this approach have been shown to out-perform both local search and backtracking on certain problems [10,18], but on others it does not achieve the scalability of local search. For further discussion on hybrids see section 6.

This paper describes a new approach called *Incomplete Dynamic Backtracking* (IDB). Inspired by DB and PDB, it is a backtracker that is able to jump back to an earlier variable without removing the assignments to intervening variables. However, it allows total flexibility in the choice of backtracking variable, which may be chosen either randomly or using any desired heuristic. It records no information about which parts of the search space have been visited, thus sacrificing completeness. The aim is (i) to maximise scalability at the expense of completeness, (ii) to exploit powerful pruning

techniques and heuristics available to backtrackers, and (iii) to avoid memory-intensive learning methods. It is hoped that the focus on pruning techniques and scalability will pay off on large structured problems that are challenging for both backtracking and local search.

Section 2 describes IDB and its integration with pruning techniques and heuristics. Section 3 evaluates it on the n-queens problem, and shows that it performs like a local search algorithm. n-queens is not intrinsically hard and was chosen partly for illustrative purposes, but in section 4 IDB is applied to a challenging optimisation problem: the construction of Golomb rulers. We take an existing constraint-based backtracking algorithm for Golomb rulers and replace its chronological backtracking by IDB. This greatly improves its scalability, and the new algorithm also out-performs a genetic algorithm. Section 5 describes an IDB algorithm for another hard optimisation problem: the construction of maximum cliques. The new algorithm is compared with a wide variety of others on standard benchmarks, and is beaten by only one. Finally, section 6 discusses relationships between IDB and other hybrid approaches.

2. Incomplete dynamic backtracking

In a constraint satisfaction problem (CSP) we are given a set of variables $\{v_1, \ldots, v_n\}$ each with a domain of values $D_i = \{V_1^i, \ldots, V_m^i\}$, and constraints C on subsets of the variables defining their permitted combinations of values. The CSP is to find an assignment $\{v_1 = V_{s_1}^1, \ldots, v_n = V_{s_n}^n\}$ that violates none of the constraints. We first describe the basic IDB schema for the CSP, then elaborate it and describe how to apply it to optimisation problems.

2.1. The basic algorithm

The basic IDB schema is shown in figure 1. A is the current set of assignments, initialised to $\{\}$. V is the current set of unassigned variables, initialised to the full set of variables $\{v_1, \ldots, v_n\}$. The integer $b \geqslant 1$ is a parameter. The algorithm proceeds by selecting random unassigned variables, and assigning values to them using a value ordering heuristic VH (discussed below). On reaching a dead-end (in which each domain value for the selected variable is inconsistent with a current assignment in A under a constraint in C) it backtracks by randomly removing b assignments from A (or fewer if $|A| < b$). Termination is not guaranteed but occurs if all variables are assigned ($V = \{\}$), in which case the set of assignments A is a solution. This algorithm is correct because no assignment is made unless it is consistent with all previous assignments. We now describe how it can be enhanced by the use of both standard and novel heuristics.

2.2. Forward checking and dynamic variable ordering

A simple and commonly-used form of constraint propagation is *forward checking*. On assigning a value to a variable, some values in the domains of currently unassigned

```
function IDB(b)
    A = {}, V = {v_1, ..., v_n}
    while V ≠ {}
        v_i = random-member(V)
        d = VH(D_i) such that v_i = d is consistent with A under C
        if (d = null) [not found]
            do min(b, |A|) times
                (v_j = d') = random-member(A)
                A = A − {v_j = d'}, V = V ∪ {v_j}
        else
            A = A ∪ {v_i = d}, V = V − {v_i}
    return A
```

Figure 1. Basic incomplete dynamic backtracking (IDB).

variables are removed. These are the values that would cause constraint violations if assigned. If the domain of an unassigned variable becomes empty then backtracking occurs. Domain sizes are useful for guiding the selection of variables for assignment, a common heuristic being to select a variable with minimum domain size.

To combine IDB with these techniques we must be able to unassign variables in any order, leaving the state of the unassigned variables as if forward checking had been applied to the currently assigned variables. To do this we need a new implementation trick. Instead of simply removing values from unassigned variable domains, a *conflict count* c_{ij} is maintained for each value j in the domain D_i of each variable v_i (assigned or not). The integer c_{ij} denotes how many constraints would be violated if the assignment $v_i = V^i_j$ were added. When $c_{ij} \neq 0$ the value V^i_j is treated as though it has been deleted from domain D_i, and it cannot be used in an assignment. Note that these are also maintained in the domains of assigned variables: for such a variable c_{ij} denotes how many constraints would be violated if the variable were *reassigned* to $v_i = V^i_j$. Now the state of any variable domain is independent of assignment order, and we can unassign variables in an arbitrary order.

The IDB schema for forward checking is shown in figure 2. Variables are selected using the minimum-domain heuristic (MD). All conflict counts are initialised to zero. The number of values in a domain with zero conflict count plays the role of domain size for MD. Values are again selected using some heuristic denoted by VH, but values are only allowed for assignment if their conflict count is zero and if propagating the assignment causes no domain wipe-out. If there is no such value then b variables are unassigned, as in the basic schema. Variables may be selected for unassignment using any heuristic BH.

Conflict counts are updated incrementally: to propagate a new assignment $v_a = V^a_k$, increment any c_{ij} such that the assignment $v_i = V^i_j$ is inconsistent with the new assignment under a constraint in C. Only constraints involving the newly assigned variable need be checked. On unassigning a variable the process is reversed. This form of

```
function IDB(b)
    A = {}, c_ij = 0, V = {v_1, ..., v_n}
    while V ≠ {}
        v_i = MD(V)
        d = VH(D_i) such that c_id = 0 and propagate(v_i = d) = true
        if (d = null) [not found]
            do min(b, |A|) times
                (v_j = d') = BH(A), A = A − {v_j = d'}, V = V ∪ {v_j}
                unpropagate(v_j = d')
        else
            A = A ∪ {v_i = d}, V = V − {v_i}
    return A

function propagate(v_i = d)
    OK = true
    for all v_j ∈ {v_1, ..., v_{i−1}, v_{i+1}, ..., v_n}
        for all d' ∈ D_j
            increment c_jd' if (v_j = d') is inconsistent with v_i = d under C
            if (for all d' ∈ D_j (c_jd' = 0)) then OK = false
    if (OK = false) unpropagate(v_i = d)
    return OK

function unpropagate(v_i = d)
    for all v_j ∈ {v_1, ..., v_{i−1}, v_{i+1}, ..., v_n}
        for all d' ∈ D_j
            decrement c_jd' if (v_j = d') is inconsistent with v_j = d under C
```

Figure 2. IDB with forward checking.

propagation is more expensive than standard forward checking, which only examines the domains of unassigned variables, but the memory requirement is the same: for n variables and m values in each domain, mn conflict counts are required. The extension of the conflict count technique to arc consistency is discussed in section 6.1.

To prove correctness we first show that any state (partial assignment) can be reached by IDB with conflict counts if and only if it can be reached by FC (standard forward checking). First, consider a state reachable by IDB. The domain of any variable (unassigned or assigned) must be non-empty, therefore all unassigned variables have non-empty domains, therefore the state is FC-reachable. Second, consider a state that is FC-reachable. No combination of its variable assignments must violate a binary constraint, therefore IDB can make the assignments without incrementing the conflict counts for the assigned values, so $c_{ij} = 0$ for each assignment $v_i = V_j^i$ in the state. Therefore domain wipe-out will not occur for any of the assigned variables. Moreover, FC-reachability implies that none of the unassigned variables has an empty domain. In

other words, the state can be recreated by IDB without emptying the domain of any variable, so it is IDB-reachable.

A solution is an example of a partial assignment, so the same set of solutions are reachable by IDB and FC. The correctness of FC is not in question so this establishes the correctness of IDB. It also supports the claim that forward checking is fully integrated with IDB: the set of partial assignments to be explored is the same (though for any given problem the set of partial assignments encountered are unlikely to be identical).

2.3. New heuristics

In the basic schema we randomly selected variables for unassignment. Given conflict counts an obvious BH heuristic is the complement of the minimum-domain heuristic: unassign the variable with the largest current domain, breaking ties randomly (recall that assigned variables can also be assigned a domain size using conflict counts). This heuristic sometimes improves performance.

Another technique sometimes used with backtracking is *value ordering*: for a given variable, values are selected for assignment in an order determined by a heuristic. The intent is to choose the value most likely to lead to a solution, an idea that can in principle be applied to IDB. We have found that a different type of value ordering heuristic denoted by VH often enhances performance. Instead of finding the best value, it assigns each variable to its last assigned value where possible, with random initial values. This speeds the rediscovery of consistent assignments to subsets of the variables. However, IDB attempts to use a different (randomly-chosen) value for one variable each time a dead-end occurs; this appears to help by introducing a little variety.

2.4. Application to optimisation problems

Given an *objective function* on CSP solutions we may wish to find a solution with minimum value under this function. Backtracking algorithms can be applied to these problems by applying them iteratively, restarting after each solution with the added constraint that any solution must be better (under the objective function). Alternatively, the search can simply continue without restarting, but with the new constraint added. These ideas have been used with systematic backtracking in Constraint Programming implementations of branch-and-bound, and they can also be applied to IDB. We restart the search after each solution, and until reaching the first dead-end we reuse assignments from the previous solution where possible.

3. Application to n-queens

We have described how to combine IDB with several standard techniques from systematic backtracking, allowing us to replace chronological backtracking by IDB in powerful algorithms. It remains to be seen whether this has any beneficial effect that justifies loss of completeness and introduction of the parameter b. We first evaluate IDB on

the well-known n-queens problem. Though fashionable several years ago n-queens is no longer considered a challenging problem. However, large instances still defeat most backtrackers and it is therefore of interest.

The problem is as follows. Consider a generalised chess board, which is a square divided into $n \times n$ smaller squares. Place n queens on it in such a way that no queen attacks any other. A queen *attacks* another if it is on the same row, column or diagonal (in which case both attack each other). We can model this problem using n variables each with domain $D_i = \{1, \ldots, n\}$. A variable v_i corresponds to a queen on row i (there is one queen per row), and the assignment $v_i = j$ denotes that the queen on row i is placed in column j, where $j \in D_i$. The constraints are $v_i \neq v_j$ and $|v_i - v_j| \neq |i - j|$ where $1 \leqslant i < j \leqslant n$. We must assign a domain value to each variable without violating any constraint.

3.1. Experimental results

Minton et al. [30] compared the performance of backtracking and local search on n-queens problems up to $n = 10^6$. They executed each algorithm 100 times for various values of n, with an upper bound of $100n$ on the number of steps (backtracks or repairs) and reported the mean number of steps and the success rate as a percentage. We reproduce the experiment up to $n = 1000$, citing their results for the Min-Conflicts local search algorithm (denoted by LS+MC) and a backtracker augmented with the Min-Conflicts heuristic (denoted by CB+MC). We compute results for chronological backtracking with random variable ordering (CB), CB with forward checking (CB+FC) and CB+FC with dynamic variable ordering based on minimum domain size (CB+FC+MD). We also obtain results for these three algorithms with CB replaced by IDB, and for two further IDB algorithms using the BH and VH heuristics described in section 2.3. The IDB parameter b is set to 1 for $n = 100$ and $n = 1000$, and 2 for $n = 10$ (these values gave the best results).

The results in table 1 show that replacing CB by IDB greatly boosts performance in three cases: the simple backtracking algorithm, backtracking with forward checking, and forward checking with dynamic variable ordering. Even the basic IDB algorithm scales better than all the CB algorithms (other than CB+MC, discussed below) and IDB+FC+MD performs like LS+MC. The new backtracking (BH) and value ordering (VH) heuristics further boost performance, making IDB the best reported algorithm in terms of backtracks; it also beats another hybrid called Weak Commitment Search [49] which requires approximately 35 steps for large n [34]. However, in terms of CPU time IDB scales more poorly than CB+FC. The time per backtrack for both scales roughly linearly with n, but we found that IDB+FC+MD takes approximately $3.6n$ μs per backtrack, while CB+FC+MD takes $0.16n$ μs (measured by performing a linear regression on mean times over 1000 runs for $n = 10$–100 in steps of 10). This clearly shows the increased expense of forward checking in IDB, but this is outweighed by its improved scalability. IDB+FC+MD and LS+MC both take a roughly constant number of steps as n increases, hence a linear time in n. We were unable to fully compare LS+MC and the

Table 1
Chronological backtracking, IDB and min-conflicts on n-queens.

Algorithm	$n = 10$	$n = 100$	$n = 1000$
CB	81.0 (100%)	9929 (1%)	—
CB+FC	25.4 (100%)	7128 (39%)	98097 (3%)
CB+FC+MD	14.7 (100%)	1268 (92%)	77060 (24%)
IDB	112 (100%)	711 (100%)	1213 (100%)
IDB+FC	33.0 (100%)	141 (100%)	211 (100%)
IDB+FC+MD	23.8 (100%)	46.3 (100%)	41.2 (100%)
IDB+FC+MD+BH	13.0 (100%)	8.7 (100%)	13.3 (100%)
IDB+FC+MD+BH+VH	12.7 (100%)	8.0 (100%)	12.3 (100%)
LS+MC	57.0 (100%)	55.6 (100%)	48.8 (100%)
CB+MC	46.8 (100%)	25.0 (100%)	30.7 (100%)

best IDB by taking n up to 1 million because IDB requires n^2 conflict counts, whereas LS+MC requires only linear memory in n. However, the results hold up to $n = 4000$.

It should be noted that IDB is not the only backtracker to perform like local search on n-queens. Similar results were obtained by Minton et al.'s CB+MC algorithm (see table 1), as well as others. Such algorithms rely on good value ordering heuristics. In CB+MC an initial total assignment I is generated by the MC heuristic and used to guide CB in two ways. Firstly, variables are selected for assignment on the basis of how many violations they cause in I. Secondly, values are tried in ascending order of number of violations with currently unassigned variables, an example of a value ordering heuristic. This *informed backtracking* algorithm performs almost identically to LS+MC on n-queens. However, CB+MC is still prone to the same drawback as most backtrackers: a poor choice of assignment high in the search tree will still take a very long time to recover from. IDB is able to modify earlier choices, as long as the b parameter is set sufficiently high, so it can recover from poor early decisions. This difference is not apparent on the n-queens problem, but will be on problems for which no good value ordering heuristic is available.

If these results extend to truly challenging combinatorial problems, then IDB is a promising generic approach: given a structured problem that is unsuitable for standard local search, yet too large to solve by systematic backtracking, IDB may be the best option. In the next two sections we evaluate IDB on hard optimisation problems.

4. Application to Golomb rulers

The Golomb Ruler Problem (GRP) has been studied for several decades. Possibly the first reference to it was in connection with radio communications [2]. Since then it has found applications in X-ray crystallography, coding theory, linear arrays of sensors and antennae, and pulse phase modulation communication [39]. A *Golomb ruler* is an ordered sequence of integers $0 = x_1 < x_1 < \cdots < x_m$ such that the $m(m-1)/2$ differences $x_j - x_i$ $(j > i)$ are distinct. The ruler is said to contain m *marks* and have

length x_m. An *optimal* Golomb ruler has minimum length. The aim may be to find an optimal ruler and verify its optimality, or simply to find a near-optimal ruler.

The GRP has several advantages as a benchmark problem for search algorithms: it is easily stated, well-studied, derived from real applications, has very few optimal solutions, and its difficulty grows rapidly with the number of marks. The 1953 paper listed optimal rulers with up to 8 marks, and subsequent papers have presented increasingly large optimal rulers. At the time of writing the largest optimal ruler found and verified has 21 marks, found by distributed processing over the internet and taking 2,467 weeks of CPU time and almost 10^{15} search tree nodes.[1] Specialised algorithms based on the theory of difference sets (for example [1]) can be used to find large, high-quality rulers (currently up to 150 marks) and most such rulers are conjectured to be optimal.

The GRP is very challenging for backtracking algorithms, and it is problem number 6 in the CSPLib benchmark library[2] (a web-based collection of constraint problems). Smith et al. [47] treated the GRP as an exercise in constraint modelling, using ILOG Solver (a commercial constraint solver) to implement and compare 15 backtracking algorithms. In experiments with up to 11 marks they found considerable variation in performance between the best and worst algorithms, demonstrating the importance of careful modelling.

Because the GRP rapidly becomes harder with problem size, stochastic search seems a promising approach. Surprisingly little work seems to have been done in this area, possibly because its optimal solutions are so sparse, but [45] used a genetic algorithm to find near-optimal rulers with up to 16 marks. When applying stochastic search to combinatorial problems, a major design decision is how constraints are to be handled. A popular method uses variations on the idea of a *penalty function*. Here the search space is the total variable assignments and the objective function is a composite of (i) a measure of distance from feasibility (for example the number of constraint violations) and (ii) the objective function specified in the original problem. This is the approach taken by Soliday et al. for their GRP genetic algorithm. Their objective function is the inverse of a polynomial in two variables: the ruler length and the number of duplicated differences.

4.1. The algorithm

The GRP presents an interesting challenge for our approach: if we take a good GRP backtracking algorithm and replace its chronological backtracking by IDB, as we did with n-queens, will its scalability improve? To test this we use Smith et al.'s backtracking algorithm based on a *ternary and binary constraint* CSP model, which gave good results. (Their best model used an all-different constraint, which we have not yet combined with IDB.) This model uses m variables $x_1, \ldots, x_m$ each with domain $\{0, \ldots, \ell\}$ where ℓ is the permitted ruler length, and is the function to be minimised. $m(m-1)/2$ auxiliary variables d_{ij} are defined for $1 \leqslant i < j \leqslant m$. Ternary constraints $d_{ij} = |x_i - x_j|$ and binary constraints $d_i \neq d_j$ $(i < j)$ are imposed. We simplify the model slightly: the x_i

[1] http://members.aol.com/golomb20/.

[2] http://www.csplib.org.

are not constrained to be ordered, nor is the symmetry-breaking constraint $d_{12} < d_{m-1,m}$ imposed. The model is therefore highly symmetrical, but this is unimportant because IDB is incomplete (see section 6 for a discussion on symmetry). A solution in standard form can easily be derived by sorting the x_i into ascending order then subtracting x_1 from each; x_m then gives the actual length of the ruler.

Smith et al. tried branching on the x_i or the d_{ij} or both, and experimented with variable orderings based either on the smallest domain or on the lexicographic ordering. Perhaps surprisingly, the lexicographic ordering gave the best results using either the x_i or d_{ij}; we use the lexicographic ordering on the x_i. To find optimal or near-optimal rulers we use the approach described in section 2.4: on finding a solution of length ℓ constraints $x_i < \ell$ $(i = 1, \ldots, m)$ are added and the search restarted. We also use conflict counts to perform forward checking, a random BH heuristic, and the VH value ordering heuristic described in section 2.3.

4.2. *Experimental results*

IDB is compared with two backtracking algorithms implemented in ILOG Solver, and with a genetic algorithm. It was executed on a 300 MHz DEC Alphaserver 1000A 5/300 under Unix, Solver on a Silicon Graphics O2, and the genetic algorithm (denoted by GA) on a 60 MHz Pentium under Linux. All IDB results used a parameter value $b = 2$, were given a large initial length (5 times greater than the known optimal length) and are medians over 100 runs. IDB execution times do not include initialisation.

Figure 3 compares IDB with two Solver algorithms: Solver(1) denotes the ternary and binary constraint algorithm on which IDB is based, and Solver(2) denotes the best of the 15 Solver algorithms, the latter using an *all-different* constraint on the d_{ij} instead of disequalities, order constraints and improved bounds on the d_{ij}. All three algorithms were executed until finding an optimal ruler.

Table 2
Comparison of the genetic algorithm and IDB on (near-)optimal rulers.

Marks	GA		IDB	
	Length	Sec	Length	Sec
5	**11**	0.05	**11**	<0.01
6	**17**	0.15	**17**	<0.01
7	**25**	0.17	**25**	<0.01
8	35	13	**34**	0.08
9	**44**	82	**44**	0.47
10	62	103	**55**	1.87
11	79	39	**72**	8.16
12	103	18	95	2.37
13	124	243	113	36.0
14	168	1.298	139	29.2
15	206	874	167	42.2
16	238	1.589	200	37.8

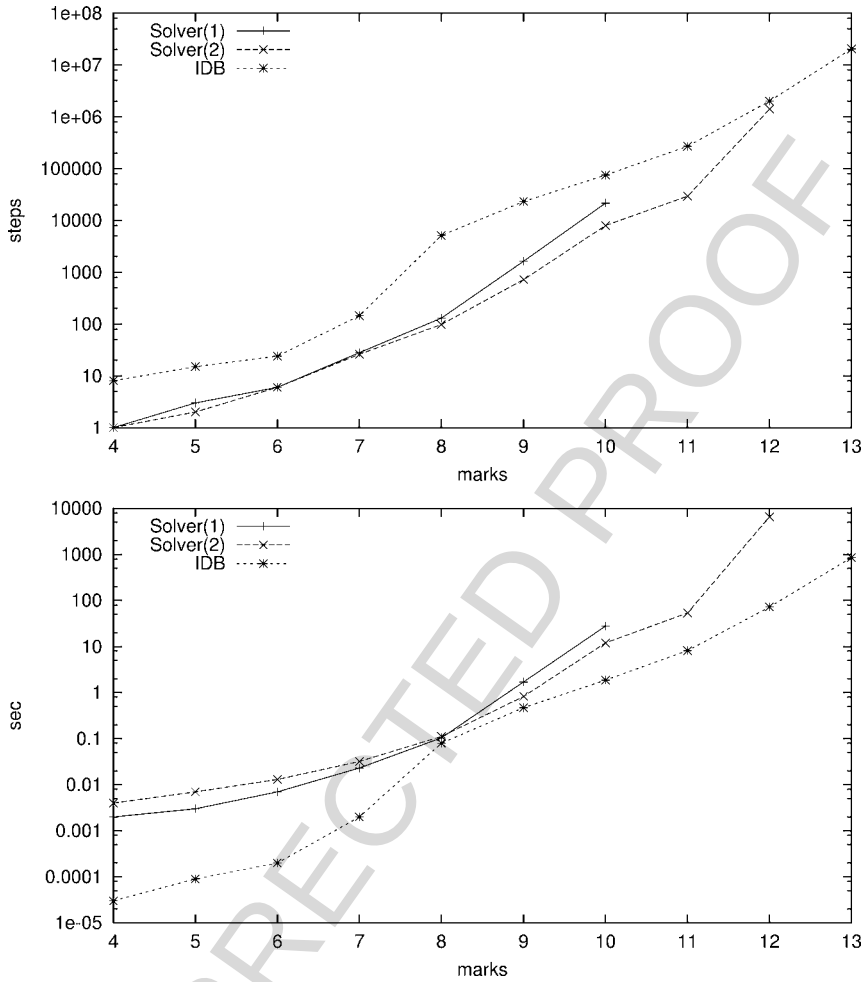

Figure 3. Comparison of ILOG Solver and IDB on optimal rulers.

The first graph shows steps (branches for Solver, backtracks for IDB) and the second CPU time in seconds. The time for Solver(2) on 12 marks is approximate and reconstructed from remarks in [47]. The times for rulers with few marks are incomparable because of Solver's initialisation times, and the algorithms were executed on different machines, but scalabilities can be compared. IDB generally makes more backtracks, but after 9 marks it shows a clear improvement, the gap in performance widening with each increase in size.]

Table 2 compares IDB with the genetic algorithm. Apart from four cases where both algorithms find optimal rulers, IDB consistently finds rulers that are closer to optimal[3] than the GA, in comparable or shorter times even when IDB's faster platform is

[3] The optimal ruler lengths for 12–16 marks are 85, 106, 127, 151, 177.

taken into account. An advantage of IDB over a GA is that it is *incremental*: backtracks are cheap, whereas a GA must calculate the fitness of each organism from scratch. But this does not seem sufficient to account for the large difference in performance, and a more likely explanation is IDB's use of pruning techniques as opposed to the genetic algorithm's use of penalty functions.

The main result is that IDB greatly improves the scalability of a powerful backtracking algorithm on a hard optimisation problem. However, this IDB algorithm cannot compete with specialised GRP algorithms. Applying it to even larger rulers we found a 25-mark ruler of length 641, a 30-mark ruler of length 1,021 and a 35-mark ruler of length 1,620. These results are poor compared to the rulers of respective lengths 480, 680 and 987 found by algorithms based on projective and affine plane construction [1], and other specialised algorithms are also faster than IDB. Nevertheless, this does not invalidate our main result.

5. Application to maximum cliques

The Maximum Clique Problem (MCP) has been the subject of four decades of research. It was one of the first problems shown to be NP-complete, and theoretical results indicate that even near-optimal solutions are hard to find. Its applications include computer vision, coding theory, tiling, fault diagnosis and the analysis of biological and archaeological data, and it provides a lower bound for the chromatic number of a graph. It was one of the three problems proposed in a DIMACS workshop [25] as a way of comparing algorithms, the other two being satisfiability and graph colouring. Many algorithms have been applied to the MCP on a common benchmark set, and its history, applicability and rich set of available results make the MCP ideal for evaluating new approaches. A recent survey of its applications, algorithms and complexity results is given in [7].

The MCP is defined as follows. A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges between vertices. Two vertices connected by an edge are said to be *adjacent*. A *clique* is a subset of V whose vertices are pairwise adjacent. A *maximum clique* is a clique of maximum cardinality. Given a graph G the problem is to find a maximum clique, or a good approximation to one.

5.1. The algorithm

We were unable to find reported results for an MCP backtracking algorithm, so we design an IDB algorithm directly. To model the problem of finding a clique of size k we define k variables $\{v_1, \ldots, v_k\}$, each representing a vertex in the clique. Each variable has domain $\{1, \ldots, n\}$ whose values correspond to vertices in G. The constraints are $(v_i \neq p \vee v_j \neq q)$ for each pair of non-adjacent vertices p and q in G and each pair of variables $v_i \neq v_j$. No vertex is adjacent to itself so constraints with $p = q$ are allowed. A further constraint is imposed as follows. A set of integers $A = \{a_1, \ldots, a_n\}$ is maintained during search, each a_i denoting how many vertices in the current clique are non-adjacent to vertex i. The integer $a = |\{a_i \in A\colon\ a_i = 0\}|$ is also maintained

and the additional constraint is $a \geqslant k$. This is a necessary (but not sufficient) condition for the existence of sufficient unused vertices to build a clique of size k, and helps to prune the search space. Note that this model is highly symmetrical because values can be permuted among the k variables. See section 6 for a discussion of symmetry.

IDB can be used to solve this problem using the techniques described in section 2 as follows. Forward checking is applied using conflict counts. The dynamic variable ordering heuristic randomly selects an unassigned variable, and the assigned variable for backtracking is also selected at random. The value ordering heuristic VH is used. The parameter b is manually tuned to each graph. To obtain increasingly large cliques the algorithm increases k on finding each solution, and restarts the search as described in section 2.4.

5.2. Experimental results

IDB is compared with several algorithms, most results being taken from the DIMACS workshop proceedings [25]. CLIQMERGE [4] uses bipartite matching to generate large cliques from small ones. MIPO [5] uses integer programming with several lift-and-project procedures. ST (Single list Tabu) [46] is a Tabu search method. SQUEEZE [8] is an algorithm for minimising general quadratic 0–1 functions, which are used to model the problems. GSD(0) (Greedy Steepest Descent) [24] and AtA (Adaptive t-Annealing) [19] are based on neural networks. The algorithm in [23] combines simulated annealing with a greedy heuristic, denoted by SA+G below. CBH (Continuous Based Heuristic) [14] uses a continuous variable formulation and a heuristic based on rounding. RB-clique [17] uses a restricted backtracking scheme that is a trade-off between clique quality and search completeness. Fleurent and Ferland [12] apply a hybrid of Tabu search and a genetic algorithm, denoted by GA+ST below. To these DIMACS results are added two more recent algorithms. Marchiori [29] combines a genetic algorithm with a heuristic algorithm to give a hybrid called HGA. Battiti and Protasi [6] use a new local search algorithm called Reactive Local Search (RLS).

The standard set of 37 benchmark graphs proposed by the DIMACS organisers are used. The largest of these has thousands of vertices and millions of edges. *Random* graphs C$n.p$ and DSJC$n.p$ have n vertices, an edge being placed between any two vertices with a fixed probability $p/10$. *Mann* graphs are clique formulations of the set covering formulation of the Steiner Triple Problem. *Brockington* graphs contain hidden cliques; a graph brockn_m contains n vertices. *Gen* graphs genn_p_m have n vertices and known hidden cliques of size m. *Hamming* graphs hammingn_m have a vertex for each n-bit word, and an edge between two vertices if and only if their words are at least a Hamming distance m apart. *Keller* graphs are based on Keller's conjecture on tilings using hypercubes. *P-hat* graphs are random graphs, modified to have wider vertex degree spread and larger clique sizes; a graph p_hat$n$-m has n vertices.

Again IDB is implemented in C and executed on a 300 MHz DEC Alphaserver 1000A 5/300 under Unix. Following DIMACS methodology all times (except those for HGA) are normalised to this machine using the `dfmax r500.5` benchmarking pro-

Table 3
Results on DIMACS clique benchmarks (part 1 of 3).

	MIPO		CLIQMERGE		SQUEEZE		RB-clique		CBH	
Problem	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
C125.9	34	257	34	4.8	34	75.2	34	8.2	34	0.16
C250.9			44	65.1			44	47.3	41	0.8
C500.9			57	256			55	143	52	5.0
C1000.9			67	1135			65	189	60	41.4
C2000.9			75	8889			74	342	66	99.6
C2000.5			16	353			16	230	15	340
C4000.5			17	1334			18	1289		
DSJC500.5			13	47.5			13	12.9	13	1.5
DSJC1000.5			15	211			15	116	14	22.8
MANN_a27			126	527	126	2554	126	2.8	121	1.9
MANN_a45			344	8611			344	14974	336	33.0
MANN_a81			1098	>10000			1097	76.1		
brock200_2	12	2994	11	3.5	12	209	12	0.07	12	0.3
brock200_4			16	6.4	17	1452	17	2.1	16	0.2
brock400_2			25	35.2			25	2.9	24	3.4
brock400_4			25	37.5			33	249	24	1.8
brock800_2			25	108			21	220	19	10.1
brock800_4			21	111			21	213	19	10.2
gen200_p0.9_44	44	71.6	42	9.5	44	7750	42	91.7	44	0.42
gen200_p0.9_55	55	1.1	55	7.7	55	2052	55	37.9	55	0.23
gen400_p0.9_55			53	63			52	593	39	4.0
gen400_p0.9_65			65	48.2			60	622	39	3.9
gen400_p0.9_75			75	50.9			74	758	45	2.2
hamming8-4	16	160	16	4.8	16	4467	16	108	10	0.29
hamming10-4			40	476			40	24.6	35	4.7
keller4	11	652	11	3.7	11	820	11	0.16	11	0.19
keller5			27	208			27	97.3	21	4.5
keller6			56	4377			54	2258		
p_hat300-1	8	5240	8	5.7	8	283	8	0.35	8	0.77
p_hat300-2			25	22.9	25	605	25	0.9	25	0.68
p_hat300-3			36	47.2	36	13742	35	46.8	36	1.3
p_hat700-1			11	28.6			11	3.3	11	10.2
p_hat700-2			44	307			44	116	44	7.0
p_hat700-3			62	398			62	1000	60	8.9
p_hat1500-1			11	128			12	56.1	11	76.5
p_hat1500-2			65	2027			64	237	63	36.0
p_hat1500-3			94	2325			93	808	94	135

gram, which takes 46.2 seconds to execute on our machine. The value of the parameter b, an upper bound on clique size, and a limit on execution time were set after a few experimental runs, as was done for several other algorithms. The clique sizes and times shown are means over 10 runs, IDB terminating on reaching the upper bound on either clique size or execution time (the time taken to read in the graph is not included). This is fairly

Table 4
Results on DIMACS clique benchmarks (part 2 of 3).

	AtA		SA+G		GSD(0)		ST		GA+T	
Problem	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
C125.9	32.4	0.004	33.4	0.012	32.3	0.0035	34	0.092	34	0.23
C250.9	38.4	0.02	41.7	0.024	39.9	0.014	44	6.1	43	3.0
C500.9	47.6	0.09	53.4	0.18	50.2	0.057	56	6.4	56	9.4
C1000.9	56.3	0.48	62.6	0.37	58.6	0.23	65	88.6	64	14.4
C2000.9	62.8	2.6	70.6	0.76	66.5	0.9	72	62.2	72.6	41.2
C2000.5	12.4	2.6	13.7	0.15	12.4	0.92	16	66.7	16	6.2
C4000.5	13.1	12.1	14.8	0.29			17	104	17	24.9
DSJC500.5	10.8	0.1	11.4	0.035	11	0.057	13	0.75	13	2.9
DSJC1000.5	11.5	0.53	12.5	0.07	12.8	0.23	14	0.23	15	4.4
MANN_a27	120.3	0.1	126	0.012	125	0.032	125	0.65	125.6	1.3
MANN_a45	332.7	0.26	343	0.094	341	0.25	342	6.1	339	195
MANN_a81	1086	8.4	1092	1			1096	213	1080	9.0
brock200_2	9.1	0.012	9.3	0.006	8.8	0.009	11	0.18	12	25.4
brock200_4	13.4	0.012	14	0.006	13.5	0.009	16	0.084	17	68.2
brock400_2	21.3	0.06	22.3	0.041	22.1	0.035	24	1.1	25	1.13
brock400_4	21.3	0.06	22.4	0.041	21.6	0.035	25	6.3	27	27.8
brock800_2	16.9	0.31	18.3	0.076	17.9	0.14	20	2.2	21	3.7
brock800_4	16.5	0.31	18.2	0.076	17.5	0.14	20	0.92	21	7.4
gen200_p0.9_44	33.6	0.02	39.2	0.024	36.8	0.008	44	2.6	44	7.0
gen200_p0.9_55	39.2	0.02	46.5	0.024	38.4	0.008	55	1.1	55	0.62
gen400_p0.9_55	40.6	0.08	49.8	0.1	46.5	0.035	52	2.5	54	11.3
gen400_p0.9_65	37.3	0.09	57.4	0.18	44.7	0.035	65	6.3	65	10.0
gen400_p0.9_75	39.2	0.1	71.1	0.28	45.8	0.035	75	2.3	75	23.2
hamming8-4	16	0.02	16	0.018	16	0.014	16	0.025	16	0.0
hamming10-4	32	0.43	38.1	0.34	35	0.24	36	0.47	40	38.0
keller4	8.4	0.01	10.8	0.012	10.2	0.006	11	0.017	11	0.0
keller5	18.5	0.18	24.3	0.29	23.1	0.14	27	6.8	27	126
keller6	37.3	3	59	1269			56	162	55.33	290
p_hat300-1	7	0.04	7	0.012	7	0.02	8	0.05	8	0.16
p_hat300-2	22.8	0.045	24.1	0.018	24.2	0.02	25	0.034	25	0.26
p_hat300-3	33.1	0.034	33.6	0.018	33.1	0.02	36	1.8	36	7.8
p_hat700-1	8.4	0.21	8.7	0.047	8	0.11	11	0.81	11	1.8
p_hat700-2	42	0.19	43.4	0.065	42.6	0.11	44	3.1	44	7.3
p_hat700-3	58.4	0.26	61.1	0.076	58.5	0.11	62	6.2	62	168
p_hat1500-1	9.4	0.91	9.8	0.11	9.9	0.53	11	0.34	11.9	14.0
p_hat1500-2	61	1	63.9	0.16	60.9	0.53	65	2.1	65	7.6
p_hat1500-3	86.2	0.89	92.2	0.19	86.2	0.52	93	5.2	93.33	48.4

typical of the experimental approaches used for the other algorithms, though there are several methodologies.

Tables 3–5 show the results split into three tables. Unavailable results are denoted by empty boxes. To compare the algorithms we count the number of graphs for which the mean clique size found by each algorithm was largest. This seems fairer than comparing the greatest clique sizes found, because it helps to eliminate the effects of lucky finds.

Table 5
Results on DIMACS clique benchmarks (part 3 of 3).

	HGA		RLS		IDB		
Problem	Size	Time*	Size	Time	Size	Time	Noise
C125.9	34	0.3	34	0.072	34	0.08	3
C250.9	42.6	3.6	44	0.097	44	0.66	6
C500.9	52.9	17.3	57	1.76	57	110	5
C1000.9	58	93.3	68	53.3	67	231	4
C2000.9	67.1	330	77.6	198	75.3	1352	4
C2000.5	14.4	61.8	16	7.92	15.5	203	2
C4000.5	15.4	307	18	524	17	401	2
DSJC500.5	12.3	3.2	13	0.115	13	1.41	2
DSJC1000.5	13.7	24.9	15	3.12	15	388	3
MANN_a27	125	3.0	126	2.17	125	2.09	1
MANN_a45	342	144	343.6	95.8	342	3.3	1
MANN_a81	1096	1149	1098	571	1095.2	1476	1
brock200_2	11.6	1.4	12	1.5	12	0.52	4
brock200_4	15.6	1.1	17	4.68	17	5.2	5
brock400_2	23.5	3.3	26.1	10.1	25	18.2	4
brock400_4	24.1	4.5	32.4	26.2	33	310	4
brock800_2	18.8	15.5	21	4.01	21	168	3
brock800_4	18.7	23.8	21	3.50	21	187	3
gen200_p0.9_44	40.7	3.0	44	0.097	44	0.26	6
gen200_p0.9_55	55	0.7	55	0.072	55	0.05	12
gen400_p0.9_55	49	13.3	55	0.459	55	50.0	4
gen400_p0.9_65	55.8	13.8	65	0.121	65	0.7	5
gen400_p0.9_75	65	7.6	75	0.145	75	0.40	5
hamming8-4	16	0.08	16	0.072	16	0.0045	1
hamming10-4	37.8	34.8	40	0.362	40	4.75	3
keller4	11	0.05	11	0.072	11	0.0059	1
keller5	26.3	10.4	27	0.362	27	19.1	4
keller6	51.4	370	59	111	57	1083	3
p_hat300-1	8	1.3	8	0.072	8	0.054	2
p_hat300-2	25	1.7	25	0.072	25	0.048	5
p_hat300-3	35.2	3.6	36	0.097	36	0.43	5
p_hat700-1	10.3	12.8	11	0.241	11	1.80	2
p_hat700-2	43.9	5.7	44	0.193	44	0.27	6
p_hat700-3	61.2	12.6	62	0.217	62	0.83	7
p_hat1500-1	10.4	30.2	12	12.2	12	893	3
p_hat1500-2	64.7	44	65	0.749	65	1.62	8
p_hat1500-3	91.4	98.4	94	0.773	94	5.85	9

* Un-normalised.

Under this measure RLS ranks first with a score of 34 out of 37, followed by IDB with 28, CLIQMERGE with 26 and RB-clique with 24. The rest receive significantly lower scores, the best being CBH with 12.

Besides sometimes generating larger cliques RLS has generally lower execution times than IDB, making it a superior MCP algorithm. It also tunes its parameters auto-

matically, whereas IDB requires the user to tune its b parameter. However, RLS does have several parameters, and though fixed values of these were sufficient for all the MCP graphs, different values may be needed for other problems. Moreover, the aims of IDB and RLS are quite different: IDB is less an algorithm than a generic architecture for combining certain techniques, while RLS is a sophisticated local search algorithm. Further experimental comparisons between the two approaches would be interesting, but the MCP is the only problem to which both have been applied.

6. Discussion

Though IDB is an incomplete version of Dynamic Backtracking it performs like a local search algorithm on the n-queens problem, whereas Dynamic Backtracking itself performs like chronological backtracking [26]. On maximum cliques IDB gave results second only to those of a sophisticated local search algorithm, beating a wide variety of other algorithms including integer programming, continuous methods, genetic algorithms, neural networks, simulated annealing and another modified backtracker. On Golomb rulers it greatly improved the scaling of a constraint-based algorithm and found better solutions than a genetic algorithm. Good results have also been found on other problems. In [37] an IDB algorithm for satisfiability was shown to scale almost exactly like a well-known local search algorithm (Walksat) on hard random 3-SAT problems. It is also able to solve SAT problems that are hard for local search [38]. In [37] IDB improved the scaling of a branch-and-bound algorithm for another hard optimisation problem (low-autocorrelation binary sequences) and is the first incomplete search algorithm to find optimal solutions. In [35] IDB with the Brélaz heuristic found improved colourings for some geometric graphs.

These results show that IDB algorithms can equal local search in scalability, and it is relevant to ask why. A possible explanation is that IDB *is* a local search algorithm, and in fact this is what we claim. It is hard to prove this claim because there is no available theory to distinguish local from other forms of search. However, consider an alternative description of the basic IDB algorithm for n-queens: place queens on rows until encountering a row on which all squares are currently under attack; remove one (or more) randomly-chosen queen; repeat until all queens are placed. This clearly qualifies as local search, and is the kind of local search algorithm that might occur to a computer scientist unused to thinking in terms of constraint violations. The more complex IDB algorithms simply add further techniques. If we view IDB as local search then the main difference between IDB and (say) the Min-Conflicts local search algorithm is the search space, the objective function to be minimised, and what constitutes a local move. Min-conflicts explores a space of total assignments, attempting to minimise the number of constraint violations by performing repairs; IDB explores a space of partial assignments that are consistent under forward checking, attempting to minimise the number of unassigned variables by performing variable assignments and unassignments. IDB is a hybrid algorithm in the sense that it performs local search in the space usually explored by backtracking, and in previous papers [37,38] algorithms based on IDB have been

called Constrained Local Search. The parameter b plays the role of a *noise parameter*, enabling it to escape local minima by making a controlled number of local moves (backtracks) that increase the value of the objective function.

As with most local search algorithms, noise must be tuned to a problem or problem class. In experiments on these and other problems we have found that the sensitivity of IDB's performance to the value of b depends strongly on the problem. We have found no short cut for the tuning process, but one pattern that seems to emerge from other experiments is that higher noise is required for more structured problems. This is in contrast to standard local search in which low noise usually works better.

The relationship between backtracking and local search, or more generally between systematic and non-systematic search, is an area of active research. In a panel discussion on systematic versus stochastic constraint satisfaction [13] it was debated which are the important properties of each class of algorithm for solving satisfiable problems, and whether properties from both classes can profitably be combined. Our results contribute to this debate by supporting the view of [16]: that the poor scaling of systematic backtracking is caused by its inflexible choice of backtracking variable. By allowing a totally flexible choice we achieve a local search standard of scalability. Our results also show that local search can profitably be combined with pruning techniques.

Other researchers have designed algorithms using backtracking techniques but with improved scalability. A hybrid of the GSAT local search algorithm and Dynamic Backtracking [16] increases the flexibility in choice of backtracking variable. However, the authors note that to achieve total flexibility while preserving completeness would require exponential memory, and they recommend a less flexible version using only polynomial memory. Local Changes [48] is a complete backtracking algorithm that uses conflict analysis to unassign variables leading to constraint violation, and a heuristic similar to VH that restores assignments after backtracking. Iterative Sampling [28] restarts a constructive search every time a dead-end is reached. Weak Commitment Search [49] builds consistent partial assignments, using the min-conflict heuristic to guide value selection. On reaching a dead-end it restarts and uses learning to avoid redundant search. Learn-SAT [40] is based on Weak Commitment Search. Bounded Backtrack Search [21] is a hybrid of Iterative Sampling and chronological backtracking, alternating a limited amount of chronological backtracking with random restarts. Gomes et al. [18] periodically restart chronological or intelligent backtracking with slightly randomised heuristics. Limited Discrepancy Search [21,22] searches the neighbourhood of a consistent partial assignment, trying neighbours in increasing order of distance from the partial assignment. It is shown theoretically, and experimentally on job shop scheduling problems, to be superior to Iterative Sampling and chronological backtracking.

There is a wide variety of other hybrid approaches. Schaerf [42] describes an algorithm that searches the space of all partial assignments, which is larger than the space searched by IDB (the *consistent* partial assignments). The objective function to be minimised includes a measure of constraint violation, whereas IDB never violates a constraint. The Path-Repair Algorithm [27] is a generalisation of Schaerf's approach that includes learning, allowing complete versions to be devised. The two-phase algo-

rithm of [51] searches a space of partial assignments, alternating backtracking search with local search. It can be tuned to different problems by spending more time in either phase. De Backer et al. [11] generate partial assignments to key variables by local search, then pass them to a constraint solver that checks consistency. Pesant and Gendreau [33] use branch-and-bound to efficiently explore local search neighbourhoods. Large Neighbourhood Search [44] performs local search and uses backtracking to test the legality of moves. Crawford [9] uses local search within a complete SAT solver to select the best branching variable.

It is impractical to compare IDB directly with all other hybrids, but some of its advantages can be stated. Firstly, it is incremental: it makes small, cheap moves in the search space. The same is true of any backtracker or standard local search algorithm, but not of all hybrids (for example Iterative Sampling). Secondly, it tightly integrates standard pruning techniques with local search, whereas some hybrids only allow a degree of cooperation between local search and constraint handling. Thirdly, it is constructive, never violating a constraint. We expect this to be an advantage when solving highly structured problems.

Finally, a note on symmetry. The problems in this paper all have symmetries which we made no effort to remove. The reason is that the use of symmetry breaking does not necessarily help local search. In fact in a recent study of the maximum clique and two other problems [36] it was shown to greatly slow down IDB and another local search algorithm. We believe that local search benefits from having as many variable assignments as possible classed as solutions. However, this should not be interpreted as evidence that IDB cannot solve problems with few solutions: on a hard optimisation problem with very few solutions, it performed better than both systematic and standard local search [37].

6.1. Future work

This paper dealt only with binary constraint networks and in future work the use of IDB with other types of constraint will be explored. It has already been combined with non-binary constraints by applying it to a SAT backtracker [37,38] but there are other interesting possibilities such as interval constraints. Another technique worth exploring is the use of clause learning to escape from local minima and reduce redundant search.

The paper was also restricted to one type of constraint propagation: forward checking. This is the cheapest and sometimes the most efficient form of constraint propagation: spending more time on constraint propagation may be more expensive than extra backtracking. However, maintaining arc consistency (MAC) during backtrack search is more efficient on many problems [41]. In MAC arc revision and variable assignment are alternated, with the details of arc revision prescribed by an underlying consistency algorithm. On assigning a value to a variable, the other values in its domain are removed and any resulting lack of support is propagated among the unassigned variables. If this results in an empty domain then backtracking occurs and support is restored. The combination of IDB and MAC is worth exploring. Here we outline a MAC-IDB algorithm but leave its proof of correctness and evaluation for future work.

On [un]assigning a variable, MAC-IDB deletes [restores] other values in its domain. It recursively propagates any changes in support to values (deleted or not) in the domains of all connected variables (assigned or not) using data structures from the AC-4 algorithm [31]: for each pair of variables (v_i, v_j) connected by a constraint, and each value $V_k^i \in D_i$, a count is defined whose value is the number of values in D_j that support V_k^i. Deletion and restoration of domain values is implemented by another type of count: the number of constraints for which a domain value is currently unsupported. A value is currently in its domain if and only if this count is zero. Using these data structures MAC-IDB can maintain arc consistency while unassigning any variable. However, the cost of maintaining arc consistency in MAC-IDB will be considerably more expensive than in MAC based on AC-4, and it remains to be seen for which problems, if any, it is worthwhile.

Acknowledgments

The Cork Constraint Computation Centre is supported by Science Foundation Ireland.

References

[1] M.D. Atkinson, N. Santoro and J. Urrutia, Integer sets with distinct sums and differences and carrier frequency assignments for nonlinear repeaters, IEEE Transactions on Communications 34 (1986) 614–617.

[2] W.C. Babcock, Intermodulation interference in radio systems, Bell Systems Technical Journal (January 1953) 63–73.

[3] A.B. Baker, The hazards of fancy backtracking, in: *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Vol. 1 (AAAI Press, 1994) pp. 288–293.

[4] E. Balas and W. Niehaus, Finding large cliques in arbitrary graphs by bipartite matching, in: [25, pp. 29–52].

[5] E. Balas, S. Ceria, G. Cornuejols and G. Pataki, Polyhedral methods for the Maximum Clique Problem, in: [25, pp 11–28].

[6] R. Battiti and M. Protasi, Reactive local search for the Maximum Clique Problem, Algorithmica 29(4) (2001) 610–637.

[7] I.M. Bomze, M. Budinich, P.M. Pardalos and M. Pelillo, The Maximum Clique Problem, in: *Handbook of Combinatorial Optimization*, Vol. 4, eds. D.-Z. Du and P.M. Pardalos (Kluwer Academic, Boston, MA, 1999).

[8] J.-M. Bourjolly, P. Gill, G. Laporte and H. Mercure, An exact quadratic 0–1 algorithm for the stable set problem, in: [25, pp. 53–74].

[9] J.M. Crawford, Solving satisfiability problems using a combination of systematic and local search, in: *Second DIMACS Challenge: Cliques, Coloring, and Satisfiability*, Rutgers University, NJ (October 1993).

[10] J.M. Crawford and A.B. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Vol. 2 (AAAI Press, 1994) pp. 1092–1097.

[11] B. De Backer, V. Furnon, P. Kilby, P. Prosser and P. Shaw, Local search in constraint programming: Application to the Vehicle Routing Problem, in: *Constraint Programming 97, Proceedings of Workshop on Industrial Constraint-Directed Scheduling* (1997).

[12] C. Fleurent and J.A. Ferland, Object-oriented implementation of heuristic search methods for Graph Coloring, Maximum Clique and Satisfiability, in: [25, pp. 619–652].
[13] E.C. Freuder, R. Dechter, M.L. Ginsberg, B. Selman and E. Tsang, Systematic versus stochastic constraint satisfaction, in: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (Morgan Kaufmann, San Mateo, CA, 1995) pp. 2027–2032.
[14] L.E. Gibbons, D.W. Hearn and P.M. Pardalos, A continuous based heuristic for the Maximum Clique Problem, in: [25, pp. 103–124].
[15] M.L. Ginsberg, Dynamic backtracking, Journal of Artificial Intelligence Research 1 (1993) 25–46.
[16] M.L. Ginsberg and D.A. McAllester, GSAT and dynamic backtracking, in: *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning* (Morgan Kaufmann, San Mateo, CA, 1994) pp. 226–237.
[17] M.K. Goldberg and R.D. Rivenburgh, Constructing cliques using restricted backtracking, in: [25, pp. 89–102].
[18] C. Gomes, B. Selman and H. Kautz, Boosting combinatorial search through randomization, in: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference* (AAAI Press/The MIT Press, 1998) pp. 431–437.
[19] T. Grossman, Applying the INN model to the Maximum Clique Problem, in: [25, pp. 125–146].
[20] J. Gu, Efficient local search for very large-scale satisfiability problems, SIGART Bulletin 3(1) (1992) 8–12.
[21] W.D. Harvey, Nonsystematic backtracking search, Ph.D. Thesis, Stanford University (1995).
[22] W.D. Harvey and M.L. Ginsberg, Limited discrepancy search, in: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (Morgan Kaufmann, San Mateo, CA, 1995) pp. 607–615.
[23] S. Homer and M. Peinado, Experiments with polynomial-time CLIQUE approximation algorithms on very large graphs, in: [25, pp. 147–168].
[24] A. Jagota, L. Sanchis and R. Ganesan, Approximately solving Maximum Clique using neural network and related heuristics, in [25, pp. 169–204].
[25] D.S. Johnson and M.A. Trick (eds.), *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26 (Amer. Math. Soc., Providence, RI, 1996).
[26] A.K. Jonsson and M.L. Ginsberg, Experimenting with new systematic and nonsystematic search techniques, in: *Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems*, Stanford, CA (1993).
[27] N. Jussienc and O. Lhomme, The path-repair algorithm, in: *Proceedings of the Workshop on Large Scale Combinatorial Optimization and Constraints*, Electronic Notes in Discrete Mathematics, Vol. 4 (1999).
[28] P. Langley, Systematic and nonsystematic search strategies, in: *Artificial Intelligence Planning Systems: Proceedings of the First International Conference* (1992).
[29] E. Marchiori, A simple heuristic based genetic algorithm for the Maximum Clique Problem, in: *Proceedings of the ACM Symposium on Applied Computing* (1998) pp. 366–373.
[30] S. Minton, M.D. Johnston, A.B. Philips and P. Laird, Minimizing conflicts: A heuristic repair method for Constraint Satisfaction and Scheduling Problems, Artificial Intelligence 58(1–3) (1992) 160–205.
[31] R. Mohr and T.C. Henderson, Arc and path consistency revisited, Artificial Intelligence 28 (1986) 225–233.
[32] P. Morris, The breakout method for escaping from local minima, in: *Proceedings of the 11th National Conference on Artificial Intelligence* (AAAI Press, 1993) pp. 40–45.
[33] G. Pesant and M. Gendreau, A view of local search in constraint programming, in: *Principles and Practice of Constraint Programming, Proceedings of the Second International Conference*, Lecture Notes in Computer Science, Vol. 1118 (Springer, Berlin, 1996) pp. 353–366.
[34] D.G. Pothos and E.B. Richards, An empirical study of min-conflict hill climbing and weak commit-

ment search, in: *Proceedings of the CP-95 Workshop on Studying and Solving Really Hard Problems* (Cassis, 1995) pp. 140–146.

[35] S.D. Prestwich, Coloration neighbourhood search with forward checking, Annals of Mathematics and Artificial Intelligence 34(4) (2002) 327–340.

[36] S.D. Prestwich, First-solution search with symmetry breaking and implied constraints, in: *Proceedings of the CP-2001 Workshop on Modelling and Problem Formulation* (2001). (Available at http://www.dcs.gla.ac.uk/~pat/cp2001/papers/.)

[37] S.D. Prestwich, A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences, in: *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Vol. 1894 (Springer, Berlin, 2000) pp. 337–352.

[38] S.D. Prestwich, Stochastic local search in constrained spaces, in: *Proceedings of Practical Applications of Constraint Technology and Logic Programming* (2000) pp. 27–39.

[39] W.T. Rankin, Optimal Golomb rulers: An exhaustive parallel search implementation, Master's Thesis, Duke University (1993).

[40] E.T. Richards and B. Richards, Non-systematic search and learning: An empirical study, in: *Principles and Practice of Constraint Programming, Proceedings of the Fourth International Conference*, Lecture Notes in Computer Science, Vol. 1520 (Springer, Berlin, 1998) pp. 370–384.

[41] D. Sabin and G. Freuder, Understanding and improving the MAC algorithm, in: *Principles and Practice of Constraint Programming, Proceedings of the Third International Conference*, Lecture Notes in Computer Science, Vol. 1330 (Springer, Berlin, 1999) pp. 167–181.

[42] A. Schaerf, Combining local search and look-ahead for Scheduling and Constraint Satisfaction problems, in: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (Morgan Kaufmann, San Mateo, CA, 1997) pp. 1254–1259.

[43] B. Selman, H. Levesque and D. Mitchell, A new method for solving hard satisfiability problems, in: *Proceedings of the 10th National Conference on Artificial Intelligence* (MIT Press, 1992) pp. 440–446.

[44] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: *Principles and Practice of Constraint Programming, Proceedings of the Fourth International Conference*, Lecture Notes in Computer Science, Vol. 1520 (Springer, Berlin, 1998) pp. 417–431.

[45] S.W. Soliday, A. Homaifar and G.L. Libby, Genetic algorithm approach to the search for Golomb rulers, in: *Proceedings of the Sixth International Conference on Genetic Algorithms*, Vol. 1 (Morgan Kaufmann, San Mateo, CA, 1995) pp. 528–535.

[46] P. Soriano and M. Gendreau, Tabu search algorithms for the Maximum Clique Problem, in: [25, pp. 221–244].

[47] B. Smith, K. Stergiou and T. Walsh, Modelling the Golomb Ruler problem, Research Report 1999.12, University of Leeds, England (June 1999). Presented at the IJCAI'99 Workshop on Non-Binary Constraints.

[48] G. Verfaillie and T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in: *Proceedings of the Twelfth National Conference on Artificial Intelligence* (AAAI Press, 1994) pp. 307–312.

[49] M. Yokoo, Weak-commitment search for solving constraint satisfaction problems, in: *Proceedings of the Twelfth National Conference on Artificial Intelligence* (AAAI Press, 1994) pp. 313–318.

[50] H. Zhang and M.E. Stickel, Implementing the Davis–Putnam method, Journal of Automated Reasoning 24(1–2) (2000) 77–296.

[51] J. Zhang and H. Zhang, Combining local search and backtracking techniques for constraint satisfaction, in: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Conference on Innovative Applications of Artificial Intelligence* (AAAI Press/The MIT Press, 1996) pp. 369–374.